Chapter 1
Introduction

Distributed database system (DDBS) technology is the union of what appear to be two diametrically opposed approaches to data processing: *database system* and *computer network* technologies. Database systems have taken us from a paradigm of data processing in which each application defined and maintained its own data (Figure 1.1) to one in which the data are defined and administered centrally (Figure 1.2). This new orientation results in *data independence*, whereby the application programs are immune to changes in the logical or physical organization of the data, and vice versa.

One of the major motivations behind the use of database systems is the desire to integrate the operational data of an enterprise and to provide centralized, thus controlled access to that data. The technology of computer networks, on the other hand, promotes a mode of work that goes against all centralization efforts. At first glance it might be difficult to understand how these two contrasting approaches can possibly be synthesized to produce a technology that is more powerful and more promising than either one alone. The key to this understanding is the realization

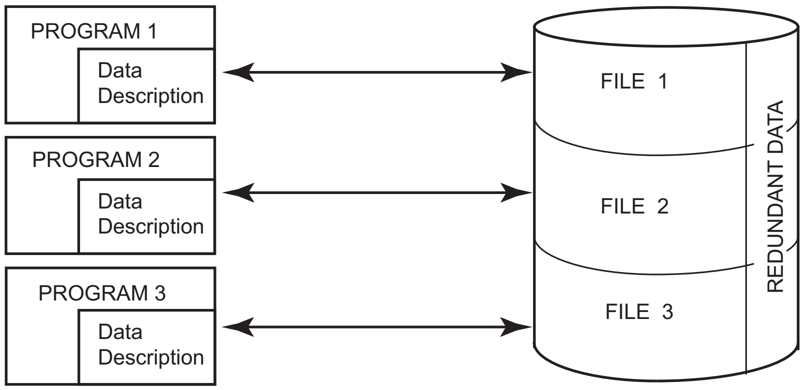

Fig. 1.1 Traditional File Processing

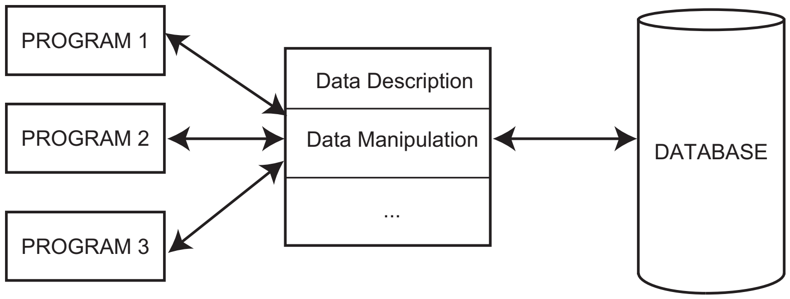

Fig. 1.2 Database Processing

that the most important objective of the database technology is *integration*, not *centralization*. It is important to realize that either one of these terms does not necessarily imply the other. It is possible to achieve integration without centralization, and that is exactly what the distributed database technology attempts to achieve.

In this chapter we define the fundamental concepts and set the framework for discussing distributed databases. We start by examining distributed systems in general in order to clarify the role of database technology within distributed data processing, and then move on to topics that are more directly related to DDBS.

1.1 Distributed Data Processing

The term *distributed processing* (or *distributed computing*) is hard to define precisely. Obviously, some degree of distributed processing goes on in any computer system, even on single-processor computers where the central processing unit (CPU) and input/output (I/O) functions are separated and overlapped. This separation and overlap can be considered as one form of distributed processing. The widespread emergence of parallel computers has further complicated the picture, since the distinction between distributed computing systems and some forms of parallel computers is rather vague.

In this book we define distributed processing in such a way that it leads to a definition of a distributed database system. The working definition we use for a *distributed computing system* states that it is a number of autonomous processing elements (not necessarily homogeneous) that are interconnected by a computer network and that cooperate in performing their assigned tasks. The "processing element" referred to in this definition is a computing device that can execute a program on its own. This definition is similar to those given in distributed systems textbooks (e.g., [Tanenbaum and van Steen, 2002] and [Colouris et al., 2001]).

A fundamental question that needs to be asked is: What is being distributed? One of the things that might be distributed is the *processing logic*. In fact, the definition of a distributed computing system given above implicitly assumes that the processing logic or processing elements are distributed. Another possible distribution is according to *function*. Various functions of a computer system could be delegated to various pieces of hardware or software. A third possible mode of distribution is according to *data*. Data used by a number of applications may be distributed to a number of processing sites. Finally, *control* can be distributed. The control of the execution of various tasks might be distributed instead of being performed by one computer system. From the viewpoint of distributed database systems, these modes of distribution are all necessary and important. In the following sections we talk about these in more detail.

Another reasonable question to ask at this point is: Why do we distribute at all? The classical answers to this question indicate that distributed processing better corresponds to the organizational structure of today's widely distributed enterprises, and that such a system is more reliable and more responsive. More importantly, many of the current applications of computer technology are inherently distributed. Web-based applications, electronic commerce business over the Internet, multimedia applications such as news-on-demand or medical imaging, manufacturing control systems are all examples of such applications.

From a more global perspective, however, it can be stated that the fundamental reason behind distributed processing is to be better able to cope with the large-scale data management problems that we face today, by using a variation of the well-known divide-and-conquer rule. If the necessary software support for distributed processing can be developed, it might be possible to solve these complicated problems simply by dividing them into smaller pieces and assigning them to different software groups, which work on different computers and produce a system that runs on multiple processing elements but can work efficiently toward the execution of a common task.

Distributed database systems should also be viewed within this framework and treated as tools that could make distributed processing easier and more efficient. It is reasonable to draw an analogy between what distributed databases might offer to the data processing world and what the database technology has already provided. There is no doubt that the development of general-purpose, adaptable, efficient distributed database systems has aided greatly in the task of developing distributed software.

1.2 What is a Distributed Database System?

We define a *distributed database* as *a collection of multiple, logically interrelated databases distributed over a computer network*. A *distributed database management system* (distributed DBMS) is then defined as *the software system that permits the management of the distributed database and makes the distribution transparent to the users*. Sometimes "distributed database system" (DDBS) is used to refer jointly to the distributed database and the distributed DBMS. The two important terms in these definitions are "*logically interrelated*" and "*distributed over a computer network*." They help eliminate certain cases that have sometimes been accepted to represent a DDBS.

A DDBS is not a "collection of files" that can be individually stored at each node of a computer network. To form a DDBS, files should not only be logically related, but there should be structured among the files, and access should be via a common interface. We should note that there has been much recent activity in providing DBMS functionality over semi-structured data that are stored in files on the Internet (such as Web pages). In light of this activity, the above requirement may seem unnecessarily strict. Nevertheless, it is important to make a distinction between a DDBS where this requirement is met, and more general distributed data management systems that provide a "DBMS-like" access to data. In various chapters of this book, we will expand our discussion to cover these more general systems.

It has sometimes been assumed that the physical distribution of data is not the most significant issue. The proponents of this view would therefore feel comfortable in labeling as a distributed database a number of (related) databases that reside in the same computer system. However, the physical distribution of data is important. It creates problems that are not encountered when the databases reside in the same computer. These difficulties are discussed in Section 1.5. Note that physical distribution does not necessarily imply that the computer systems be geographically far apart; they could actually be in the same room. It simply implies that the communication between them is done over a network instead of through shared memory or shared disk (as would be the case with *multiprocessor systems*), with the network as the only shared resource.

This suggests that multiprocessor systems should not be considered as DDBSs. Although shared-nothing multiprocessors, where each processor node has its own primary and secondary memory, and may also have its own peripherals, are quite similar to the distributed environment that we focus on, there are differences. The fundamental difference is the mode of operation. A multiprocessor system design is rather symmetrical, consisting of a number of identical processor and memory components, and controlled by one or more copies of the same operating system that is responsible for a strict control of the task assignment to each processor. This is not true in distributed computing systems, where heterogeneity of the operating system as well as the hardware is quite common. Database systems that run over multiprocessor systems are called *parallel database systems* and are discussed in Chapter 14.

A DDBS is also not a system where, despite the existence of a network, the database resides at only one node of the network (Figure 1.3). In this case, the problems of database management are no different than the problems encountered in a centralized database environment (shortly, we will discuss client/server DBMSs which relax this requirement to a certain extent). The database is centrally managed by one computer system (site 2 in Figure 1.3) and all the requests are routed to that site. The only additional consideration has to do with transmission delays. It is obvious that the existence of a computer network or a collection of "files" is not sufficient to form a distributed database system. What we are interested in is an environment where data are distributed among a number of sites (Figure 1.4).

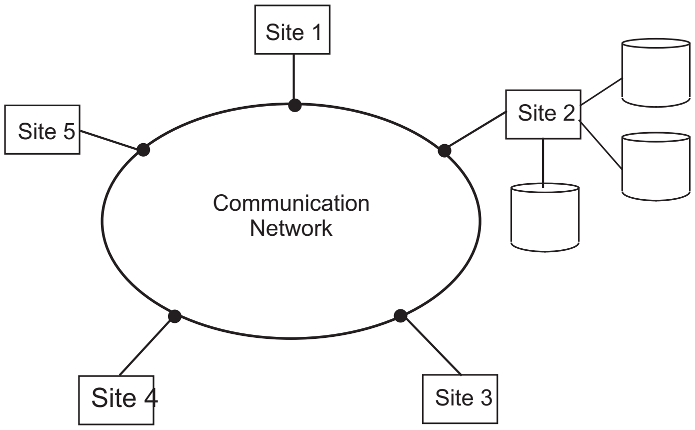

Fig. 1.3 Central Database on a Network

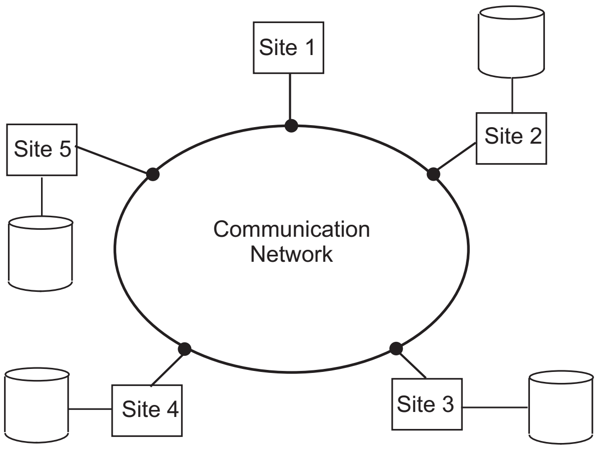

Fig. 1.4 DDBS Environment

1.3 Data Delivery Alternatives

In distributed databases, data are "delivered" from the sites where they are stored to where the query is posed. We characterize the data delivery alternatives along three orthogonal dimensions: *delivery modes*, *frequency* and *communication methods*. The combinations of alternatives along each of these dimensions (that we discuss next) provide a rich design space.

The alternative delivery modes are pull-only, push-only and hybrid. In the *pull-only* mode of data delivery, the transfer of data from servers to clients is initiated by a client pull. When a client request is received at a server, the server responds by locating the requested information. The main characteristic of pull-based delivery is that the arrival of new data items or updates to existing data items are carried out at a server without notification to clients unless clients explicitly poll the server. Also, in pull-based mode, servers must be interrupted continuously to deal with requests from clients. Furthermore, the information that clients can obtain from a server is limited to when and what clients know to ask for. Conventional DBMSs offer primarily pull-based data delivery.

In the *push-only* mode of data delivery, the transfer of data from servers to clients is initiated by a server push in the absence of any specific request from clients. The main difficulty of the push-based approach is in deciding which data would be of common interest, and when to send them to clients – alternatives are periodic, irregular, or conditional. Thus, the usefulness of server push depends heavily upon the accuracy of a server to predict the needs of clients. In push-based mode, servers disseminate information to either an unbounded set of clients (random broadcast) who can listen to a medium or selective set of clients (multicast), who belong to some categories of recipients that may receive the data.

The hybrid mode of data delivery combines the client-pull and server-push mechanisms. The continuous (or continual) query approach (e.g., [Liu et al., 1996],[Terry et al., 1992],[Chen et al., 2000],[Pandey et al., 2003]) presents one possible way of combining the pull and push modes: namely, the transfer of information from servers to clients is first initiated by a client pull (by posing the query), and the subsequent transfer of updated information to clients is initiated by a server push.

There are three typical frequency measurements that can be used to classify the regularity of data delivery. They are *periodic*, *conditional*, and *ad-hoc* or *irregular*.

In periodic delivery, data are sent from the server to clients at regular intervals. The intervals can be defined by system default or by clients using their profiles. Both pull and push can be performed in periodic fashion. Periodic delivery is carried out on a regular and pre-specified repeating schedule. A client request for IBM's stock price every week is an example of a periodic pull. An example of periodic push is when an application can send out stock price listing on a regular basis, say every morning. Periodic push is particularly useful for situations in which clients might not be available at all times, or might be unable to react to what has been sent, such as in the mobile setting where clients can become disconnected.

In conditional delivery, data are sent from servers whenever certain conditions installed by clients in their profiles are satisfied. Such conditions can be as simple as a given time span or as complicated as event-condition-action rules. Conditional delivery is mostly used in the hybrid or push-only delivery systems. Using conditional push, data are sent out according to a pre-specified condition, rather than any particular repeating schedule. An application that sends out stock prices only when they change is an example of conditional push. An application that sends out a balance statement only when the total balance is 5% below the pre-defined balance threshold is an example of hybrid conditional push. Conditional push assumes that changes are critical to the clients, and that clients are always listening and need to respond to what is being sent. Hybrid conditional push further assumes that missing some update information is not crucial to the clients.

Ad-hoc delivery is irregular and is performed mostly in a pure pull-based system. Data are pulled from servers to clients in an ad-hoc fashion whenever clients request it. In contrast, periodic pull arises when a client uses polling to obtain data from servers based on a regular period (schedule).

The third component of the design space of information delivery alternatives is the communication method. These methods determine the various ways in which servers and clients communicate for delivering information to clients. The alternatives are *unicast* and *one-to-many*. In unicast, the communication from a server to a client is one-to-one: the server sends data to one client using a particular delivery mode with some frequency. In one-to-many, as the name implies, the server sends data to a number of clients. Note that we are not referring here to a specific protocol; one-to-many communication may use a multicast or broadcast protocol.

We should note that this characterization is subject to considerable debate. It is not clear that every point in the design space is meaningful. Furthermore, specification of alternatives such as conditional **and** periodic (which may make sense) is difficult. However, it serves as a first-order characterization of the complexity of emerging distributed data management systems. For the most part, in this book, we are concerned with pull-only, ad hoc data delivery systems, although examples of other approaches are discussed in some chapters.

1.4 Promises of DDBSs

Many advantages of DDBSs have been cited in literature, ranging from sociological reasons for decentralization [D'Oliviera, 1977] to better economics. All of these can be distilled to four fundamentals which may also be viewed as promises of DDBS technology: transparent management of distributed and replicated data, reliable access to data through distributed transactions, improved performance, and easier system expansion. In this section we discuss these promises and, in the process, introduce many of the concepts that we will study in subsequent chapters.

1.4.1 Transparent Management of Distributed and Replicated Data

Transparency refers to separation of the higher-level semantics of a system from lower-level implementation issues. In other words, a transparent system "hides" the implementation details from users. The advantage of a fully transparent DBMS is the high level of support that it provides for the development of complex applications. It is obvious that we would like to make all DBMSs (centralized or distributed) fully transparent.

Let us start our discussion with an example. Consider an engineering firm that has offices in Boston, Waterloo, Paris and San Francisco. They run projects at each of these sites and would like to maintain a database of their employees, the projects and other related data. Assuming that the database is relational, we can store this information in two relations: EMP(ENO, ENAME, TITLE)[1] and PROJ(PNO, PNAME, BUDGET). We also introduce a third relation to store salary information: SAL(TITLE, AMT) and a fourth relation ASG which indicates which employees have been assigned to which projects for what duration with what responsibility: ASG(ENO, PNO, RESP, DUR). If all of this data were stored in a centralized DBMS, and we wanted to find out the names and employees who worked on a project for more than 12 months, we would specify this using the following SQL query:

```
SELECT  ENAME, AMT
FROM    EMP, ASG, SAL
WHERE   ASG.DUR > 12
AND     EMP.ENO = ASG.ENO
AND     SAL.TITLE = EMP.TITLE
```

However, given the distributed nature of this firm's business, it is preferable, under these circumstances, to localize data such that data about the employees in Waterloo office are stored in Waterloo, those in the Boston office are stored in Boston, and so forth. The same applies to the project and salary information. Thus, what we are engaged in is a process where we partition each of the relations and store each partition at a different site. This is known as *fragmentation* and we discuss it further below and in detail in Chapter 3.

Furthermore, it may be preferable to duplicate some of this data at other sites for performance and reliability reasons. The result is a distributed database which is fragmented and replicated (Figure 1.5). Fully transparent access means that the users can still pose the query as specified above, without paying any attention to the fragmentation, location, or replication of data, and let the system worry about resolving these issues.

For a system to adequately deal with this type of query over a distributed, fragmented and replicated database, it needs to be able to deal with a number of different types of transparencies. We discuss these in this section.

1.4.1.1 Data Independence

Data independence is a fundamental form of transparency that we look for within a DBMS. It is also the only type that is important within the context of a centralized DBMS. It refers to the immunity of user applications to changes in the definition and organization of data, and vice versa.

As is well-known, data definition occurs at two levels. At one level the logical structure of the data are specified, and at the other level its physical structure. The former is commonly known as the *schema definition*, whereas the latter is referred to as the *physical data description*. We can therefore talk about two types of data

[1] We discuss relational systems in Chapter 2 (Section 2.1) where we develop this example further. For the time being, it is sufficient to note that this nomenclature indicates that we have just defined a relation with three attributes: ENO (which is the key, identified by underlining), ENAME and TITLE.

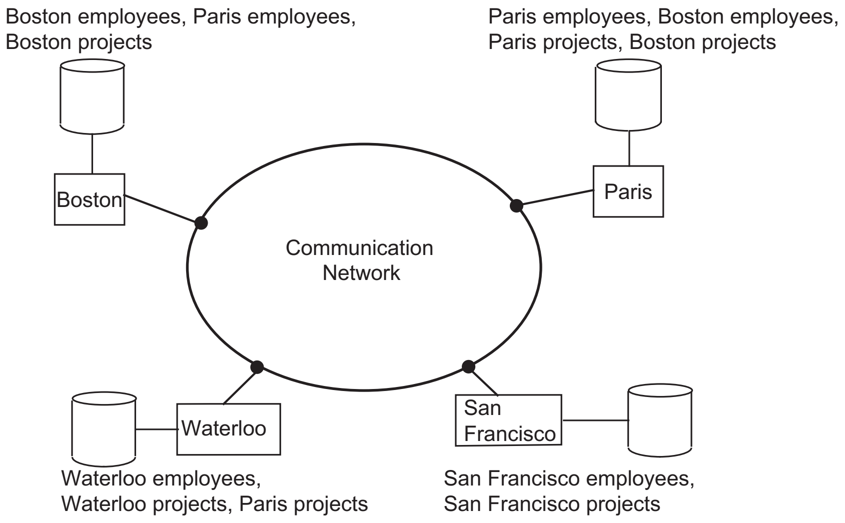
Boston employees, Paris employees,
Boston projects
Paris employees, Boston employees,
Paris projects, Boston projects
Boston
Paris
Communication
Network
Waterloo
San
Francisco
Waterloo employees,
Waterloo projects, Paris projects
San Francisco employees,
San Francisco projects

Fig. 1.5 A Distributed Application

independence: logical data independence and physical data independence. *Logical data independence* refers to the immunity of user applications to changes in the logical structure (i.e., schema) of the database. *Physical data independence*, on the other hand, deals with hiding the details of the storage structure from user applications. When a user application is written, it should not be concerned with the details of physical data organization. Therefore, the user application should not need to be modified when data organization changes occur due to performance considerations.

1.4.1.2 Network Transparency

In centralized database systems, the only available resource that needs to be shielded from the user is the data (i.e., the storage system). In a distributed database environment, however, there is a second resource that needs to be managed in much the same manner: the network. Preferably, the user should be protected from the operational details of the network; possibly even hiding the existence of the network. Then there would be no difference between database applications that would run on a centralized database and those that would run on a distributed database. This type of transparency is referred to as *network transparency* or *distribution transparency*.

One can consider network transparency from the viewpoint of either the services provided or the data. From the former perspective, it is desirable to have a uniform means by which services are accessed. From a DBMS perspective, distribution transparency requires that users do not have to specify where data are located.

Sometimes two types of distribution transparency are identified: location transparency and naming transparency. *Location transparency* refers to the fact that the command used to perform a task is independent of both the location of the data and the system on which an operation is carried out. *Naming transparency* means that a unique name is provided for each object in the database. In the absence of naming transparency, users are required to embed the location name (or an identifier) as part of the object name.

1.4.1.3 Replication Transparency

The issue of replicating data within a distributed database is introduced in Chapter 3 and discussed in detail in Chapter 13. At this point, let us just mention that for performance, reliability, and availability reasons, it is usually desirable to be able to distribute data in a replicated fashion across the machines on a network. Such replication helps performance since diverse and conflicting user requirements can be more easily accommodated. For example, data that are commonly accessed by one user can be placed on that user's local machine as well as on the machine of another user with the same access requirements. This increases the locality of reference. Furthermore, if one of the machines fails, a copy of the data are still available on another machine on the network. Of course, this is a very simple-minded description of the situation. In fact, the decision as to whether to replicate or not, and how many copies of any database object to have, depends to a considerable degree on user applications. We will discuss these in later chapters.

Assuming that data are replicated, the transparency issue is whether the users should be aware of the existence of copies or whether the system should handle the management of copies and the user should act as if there is a single copy of the data (note that we are not referring to the placement of copies, only their existence). From a user's perspective the answer is obvious. It is preferable not to be involved with handling copies and having to specify the fact that a certain action can and/or should be taken on multiple copies. From a systems point of view, however, the answer is not that simple. As we will see in Chapter 11, when the responsibility of specifying that an action needs to be executed on multiple copies is delegated to the user, it makes transaction management simpler for distributed DBMSs. On the other hand, doing so inevitably results in the loss of some flexibility. It is not the system that decides whether or not to have copies and how many copies to have, but the user application. Any change in these decisions because of various considerations definitely affects the user application and, therefore, reduces data independence considerably. Given these considerations, it is desirable that replication transparency be provided as a standard feature of DBMSs. Remember that replication transparency refers only to the existence of replicas, not to their actual location. Note also that distributing these replicas across the network in a transparent manner is the domain of network transparency.

1.4.1.4 Fragmentation Transparency

The final form of transparency that needs to be addressed within the context of a distributed database system is that of fragmentation transparency. In Chapter 3 we discuss and justify the fact that it is commonly desirable to divide each database relation into smaller fragments and treat each fragment as a separate database object (i.e., another relation). This is commonly done for reasons of performance, availability, and reliability. Furthermore, fragmentation can reduce the negative effects of replication. Each replica is not the full relation but only a subset of it; thus less space is required and fewer data items need be managed.

There are two general types of fragmentation alternatives. In one case, called *horizontal fragmentation*, a relation is partitioned into a set of sub-relations each of which have a subset of the tuples (rows) of the original relation. The second alternative is *vertical fragmentation* where each sub-relation is defined on a subset of the attributes (columns) of the original relation.

When database objects are fragmented, we have to deal with the problem of handling user queries that are specified on entire relations but have to be executed on subrelations. In other words, the issue is one of finding a query processing strategy based on the fragments rather than the relations, even though the queries are specified on the latter. Typically, this requires a translation from what is called a *global query* to several *fragment queries*. Since the fundamental issue of dealing with fragmentation transparency is one of query processing, we defer the discussion of techniques by which this translation can be performed until Chapter 7.

1.4.1.5 Who Should Provide Transparency?

In previous sections we discussed various possible forms of transparency within a distributed computing environment. Obviously, to provide easy and efficient access by novice users to the services of the DBMS, one would want to have full transparency, involving all the various types that we discussed. Nevertheless, the level of transparency is inevitably a compromise between ease of use and the difficulty and overhead cost of providing high levels of transparency. For example, Gray argues that full transparency makes the management of distributed data very difficult and claims that "applications coded with transparent access to geographically distributed databases have: poor manageability, poor modularity, and poor message performance" [Gray, 1989]. He proposes a remote procedure call mechanism between the requestor users and the server DBMSs whereby the users would direct their queries to a specific DBMS. This is indeed the approach commonly taken by client/server systems that we discuss shortly.

What has not yet been discussed is who is responsible for providing these services. It is possible to identify three distinct layers at which the transparency services can be provided. It is quite common to treat these as mutually exclusive means of providing the service, although it is more appropriate to view them as complementary.

We could leave the responsibility of providing transparent access to data resources to the access layer. The transparency features can be built into the user language, which then translates the requested services into required operations. In other words, the compiler or the interpreter takes over the task and no transparent service is provided to the implementer of the compiler or the interpreter.

The second layer at which transparency can be provided is the operating system level. State-of-the-art operating systems provide some level of transparency to system users. For example, the device drivers within the operating system handle the details of getting each piece of peripheral equipment to do what is requested. The typical computer user, or even an application programmer, does not normally write device drivers to interact with individual peripheral equipment; that operation is transparent to the user.

Providing transparent access to resources at the operating system level can obviously be extended to the distributed environment, where the management of the network resource is taken over by the distributed operating system or the middleware if the distributed DBMS is implemented over one. There are two potential problems with this approach. The first is that not all commercially available distributed operating systems provide a reasonable level of transparency in network management. The second problem is that some applications do not wish to be shielded from the details of distribution and need to access them for specific performance tuning.

The third layer at which transparency can be supported is within the DBMS. The transparency and support for database functions provided to the DBMS designers by an underlying operating system is generally minimal and typically limited to very fundamental operations for performing certain tasks. It is the responsibility of the DBMS to make all the necessary translations from the operating system to the higher-level user interface. This mode of operation is the most common method today. There are, however, various problems associated with leaving the task of providing full transparency to the DBMS. These have to do with the interaction of the operating system with the distributed DBMS and are discussed throughout this book.

A hierarchy of these transparencies is shown in Figure 1.6. It is not always easy to delineate clearly the levels of transparency, but such a figure serves an important instructional purpose even if it is not fully correct. To complete the picture we have added a "language transparency" layer, although it is not discussed in this chapter. With this generic layer, users have high-level access to the data (e.g., fourth-generation languages, graphical user interfaces, natural language access).

1.4.2 Reliability Through Distributed Transactions

Distributed DBMSs are intended to improve reliability since they have replicated components and, thereby eliminate single points of failure. The failure of a single site, or the failure of a communication link which makes one or more sites unreachable, is not sufficient to bring down the entire system. In the case of a distributed database, this means that some of the data may be unreachable, but with proper care, users

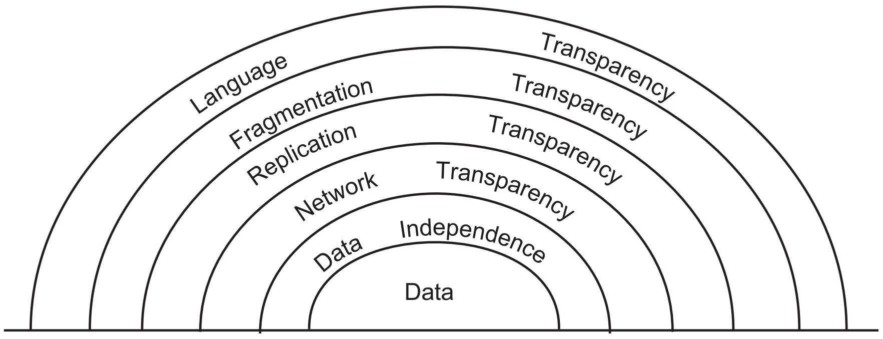

Fig. 1.6 Layers of Transparency

may be permitted to access other parts of the distributed database. The "proper care" comes in the form of support for distributed transactions and application protocols.

We discuss transactions and transaction processing in detail in Chapters 10–12. A *transaction* is a basic unit of consistent and reliable computing, consisting of a sequence of database operations executed as an atomic action. It transforms a consistent database state to another consistent database state even when a number of such transactions are executed concurrently (sometimes called *concurrency transparency*), and even when failures occur (also called *failure atomicity*). Therefore, a DBMS that provides full transaction support guarantees that concurrent execution of user transactions will not violate database consistency in the face of system failures as long as each transaction is correct, i.e., obeys the integrity rules specified on the database.

Let us give an example of a transaction based on the engineering firm example that we introduced earlier. Assume that there is an application that updates the salaries of all the employees by 10%. It is desirable to encapsulate the query (or the program code) that accomplishes this task within transaction boundaries. For example, if a system failure occurs half-way through the execution of this program, we would like the DBMS to be able to determine, upon recovery, where it left off and continue with its operation (or start all over again). This is the topic of failure atomicity. Alternatively, if some other user runs a query calculating the average salaries of the employees in this firm while the original update action is going on, the calculated result will be in error. Therefore we would like the system to be able to synchronize the *concurrent* execution of these two programs. To encapsulate a query (or a program code) within transactional boundaries, it is sufficient to declare the begin of the transaction and its end:

```
Begin_transaction SALARY_UPDATE
begin
  EXEC SQL UPDATE   PAY
           SET      SAL = SAL*1.1
end.
```

Distributed transactions execute at a number of sites at which they access the local database. The above transaction, for example, will execute in Boston, Waterloo, Paris and San Francisco since the data are distributed at these sites. With full support for distributed transactions, user applications can access a single logical image of the database and rely on the distributed DBMS to ensure that their requests will be executed correctly no matter what happens in the system. “Correctly” means that user applications do not need to be concerned with coordinating their accesses to individual local databases nor do they need to worry about the possibility of site or communication link failures during the execution of their transactions. This illustrates the link between distributed transactions and transparency, since both involve issues related to distributed naming and directory management, among other things.

Providing transaction support requires the implementation of distributed concurrency control (Chapter 11) and distributed reliability (Chapter 12) protocols — in particular, two-phase commit (2PC) and distributed recovery protocols — which are significantly more complicated than their centralized counterparts. Supporting replicas requires the implementation of replica control protocols that enforce a specified semantics of accessing them (Chapter 13).

1.4.3 Improved Performance

The case for the improved performance of distributed DBMSs is typically made based on two points. First, a distributed DBMS fragments the conceptual database, enabling data to be stored in close proximity to its points of use (also called *data localization*). This has two potential advantages:

1. Since each site handles only a portion of the database, contention for CPU and I/O services is not as severe as for centralized databases.
2. Localization reduces remote access delays that are usually involved in wide area networks (for example, the minimum round-trip message propagation delay in satellite-based systems is about 1 second).

Most distributed DBMSs are structured to gain maximum benefit from data localization. Full benefits of reduced contention and reduced communication overhead can be obtained only by a proper fragmentation and distribution of the database.

This point relates to the overhead of distributed computing if the data have to reside at remote sites and one has to access it by remote communication. The argument is that it is better, in these circumstances, to distribute the data management functionality to where the data are located rather than moving large amounts of data. This has lately become a topic of contention. Some argue that with the widespread use of high-speed, high-capacity networks, distributing data and data management functions no longer makes sense and that it may be much simpler to store data at a central site and access it (by downloading) over high-speed networks. This argument, while appealing, misses the point of distributed databases. First of all, in most of today's applications, data are distributed; what may be open for debate is how and where we process it. Second, and more important, point is that this argument does not distinguish between bandwidth (the capacity of the computer links) and latency (how long it takes for data to be transmitted). Latency is inherent in the distributed environments and there are physical limits to how fast we can send data over computer networks. As indicated above, for example, satellite links take about half-a-second to transmit data between two ground stations. This is a function of the distance of the satellites from the earth and there is nothing that we can do to improve that performance. For some applications, this might constitute an unacceptable delay.

The second case point is that the inherent parallelism of distributed systems may be exploited for inter-query and intra-query parallelism. Inter-query parallelism results from the ability to execute multiple queries at the same time while intra-query parallelism is achieved by breaking up a single query into a number of subqueries each of which is executed at a different site, accessing a different part of the distributed database.

If the user access to the distributed database consisted only of querying (i.e., read-only access), then provision of inter-query and intra-query parallelism would imply that as much of the database as possible should be replicated. However, since most database accesses are not read-only, the mixing of read and update operations requires the implementation of elaborate concurrency control and commit protocols.

1.4.4 Easier System Expansion

In a distributed environment, it is much easier to accommodate increasing database sizes. Major system overhauls are seldom necessary; expansion can usually be handled by adding processing and storage power to the network. Obviously, it may not be possible to obtain a linear increase in "power," since this also depends on the overhead of distribution. However, significant improvements are still possible.

One aspect of easier system expansion is economics. It normally costs much less to put together a system of "smaller" computers with the equivalent power of a single big machine. In earlier times, it was commonly believed that it would be possible to purchase a fourfold powerful computer if one spent twice as much. This was known as Grosh's law. With the advent of microcomputers and workstations, and their price/performance characteristics, this law is considered invalid.

This should not be interpreted to mean that mainframes are dead; this is not the point that we are making here. Indeed, in recent years, we have observed a resurgence in the world-wide sale of mainframes. The point is that for many applications, it is more economical to put together a distributed computer system (whether composed of mainframes or workstations) with sufficient power than it is to establish a single, centralized system to run these tasks. In fact, the latter may not even be feasible these days.

1.5 Complications Introduced by Distribution

The problems encountered in database systems take on additional complexity in a distributed environment, even though the basic underlying principles are the same. Furthermore, this additional complexity gives rise to new problems influenced mainly by three factors.

First, data may be replicated in a distributed environment. A distributed database can be designed so that the entire database, or portions of it, reside at different sites of a computer network. It is not essential that every site on the network contain the database; it is only essential that there be more than one site where the database resides. The possible duplication of data items is mainly due to reliability and efficiency considerations. Consequently, the distributed database system is responsible for (1) choosing one of the stored copies of the requested data for access in case of retrievals, and (2) making sure that the effect of an update is reflected on each and every copy of that data item.

Second, if some sites fail (e.g., by either hardware or software malfunction), or if some communication links fail (making some of the sites unreachable) while an update is being executed, the system must make sure that the effects will be reflected on the data residing at the failing or unreachable sites as soon as the system can recover from the failure.

The third point is that since each site cannot have instantaneous information on the actions currently being carried out at the other sites, the synchronization of transactions on multiple sites is considerably harder than for a centralized system.

These difficulties point to a number of potential problems with distributed DBMSs. These are the inherent complexity of building distributed applications, increased cost of replicating resources, and, more importantly, managing distribution, the devolution of control to many centers and the difficulty of reaching agreements, and the exacerbated security concerns (the secure communication channel problem). These are well-known problems in distributed systems in general, and, in this book, we discuss their manifestations within the context of distributed DBMS and how they can be addressed.

1.6 Design Issues

In Section 1.4, we discussed the promises of distributed DBMS technology, highlighting the challenges that need to be overcome in order to realize them. In this section we build on this discussion by presenting the design issues that arise in building a distributed DBMS. These issues will occupy much of the remainder of this book.

1.6.1 Distributed Database Design

The question that is being addressed is how the database and the applications that run against it should be placed across the sites. There are two basic alternatives to placing data: *partitioned* (or *non-replicated*) and *replicated*. In the partitioned scheme the database is divided into a number of disjoint partitions each of which is placed at a different site. Replicated designs can be either *fully replicated* (also called *fully duplicated*) where the entire database is stored at each site, or *partially replicated* (or *partially duplicated*) where each partition of the database is stored at more than one site, but not at all the sites. The two fundamental design issues are *fragmentation*, the separation of the database into partitions called *fragments*, and *distribution*, the optimum distribution of fragments.

The research in this area mostly involves mathematical programming in order to minimize the combined cost of storing the database, processing transactions against it, and message communication among sites. The general problem is NP-hard. Therefore, the proposed solutions are based on heuristics. Distributed database design is the topic of Chapter 3.

1.6.2 Distributed Directory Management

A directory contains information (such as descriptions and locations) about data items in the database. Problems related to directory management are similar in nature to the database placement problem discussed in the preceding section. A directory may be global to the entire DDBS or local to each site; it can be centralized at one site or distributed over several sites; there can be a single copy or multiple copies. We briefly discuss these issues in Chapter 3.

1.6.3 Distributed Query Processing

Query processing deals with designing algorithms that analyze queries and convert them into a series of data manipulation operations. The problem is how to decide on a strategy for executing each query over the network in the most cost-effective way, however cost is defined. The factors to be considered are the distribution of data, communication costs, and lack of sufficient locally-available information. The objective is to optimize where the inherent parallelism is used to improve the performance of executing the transaction, subject to the above-mentioned constraints. The problem is NP-hard in nature, and the approaches are usually heuristic. Distributed query processing is discussed in detail in Chapter 6 - 8.

1.6.4 Distributed Concurrency Control

Concurrency control involves the synchronization of accesses to the distributed database, such that the integrity of the database is maintained. It is, without any doubt, one of the most extensively studied problems in the DDBS field. The concurrency control problem in a distributed context is somewhat different than in a centralized framework. One not only has to worry about the integrity of a single database, but also about the consistency of multiple copies of the database. The condition that requires all the values of multiple copies of every data item to converge to the same value is called *mutual consistency*.

The alternative solutions are too numerous to discuss here, so we examine them in detail in Chapter 11. Let us only mention that the two general classes are *pessimistic* , synchronizing the execution of user requests before the execution starts, and *optimistic*, executing the requests and then checking if the execution has compromised the consistency of the database. Two fundamental primitives that can be used with both approaches are *locking*, which is based on the mutual exclusion of accesses to data items, and *timestamping*, where the transaction executions are ordered based on timestamps. There are variations of these schemes as well as hybrid algorithms that attempt to combine the two basic mechanisms.

1.6.5 Distributed Deadlock Management

The deadlock problem in DDBSs is similar in nature to that encountered in operating systems. The competition among users for access to a set of resources (data, in this case) can result in a deadlock if the synchronization mechanism is based on locking. The well-known alternatives of prevention, avoidance, and detection/recovery also apply to DDBSs. Deadlock management is covered in Chapter 11.

1.6.6 Reliability of Distributed DBMS

We mentioned earlier that one of the potential advantages of distributed systems is improved reliability and availability. This, however, is not a feature that comes automatically. It is important that mechanisms be provided to ensure the consistency of the database as well as to detect failures and recover from them. The implication for DDBSs is that when a failure occurs and various sites become either inoperable or inaccessible, the databases at the operational sites remain consistent and up to date. Furthermore, when the computer system or network recovers from the failure, the DDBSs should be able to recover and bring the databases at the failed sites up-to-date. This may be especially difficult in the case of network partitioning, where the sites are divided into two or more groups with no communication among them. Distributed reliability protocols are the topic of Chapter 12.

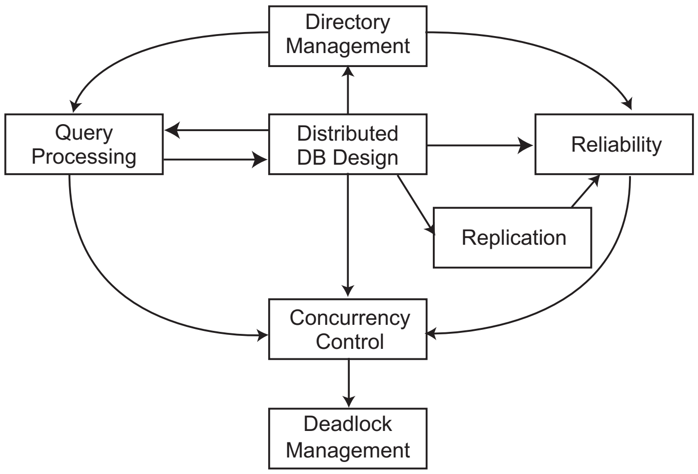

Fig. 1.7 Relationship Among Research Issues

1.6.7 Replication

If the distributed database is (partially or fully) replicated, it is necessary to implement protocols that ensure the consistency of the replicas,i.e., copies of the same data item have the same value. These protocols can be *eager* in that they force the updates to be applied to all the replicas before the transaction completes, or they may be *lazy* so that the transaction updates one copy (called the *master*) from which updates are propagated to the others after the transaction completes. We discuss replication protocols in Chapter 13.

1.6.8 Relationship among Problems

Naturally, these problems are not isolated from one another. Each problem is affected by the solutions found for the others, and in turn affects the set of feasible solutions for them. In this section we discuss how they are related.

The relationship among the components is shown in Figure 1.7. The design of distributed databases affects many areas. It affects directory management, because the definition of fragments and their placement determine the contents of the directory (or directories) as well as the strategies that may be employed to manage them. The same information (i.e., fragment structure and placement) is used by the query processor to determine the query evaluation strategy. On the other hand, the access and usage patterns that are determined by the query processor are used as inputs to the data distribution and fragmentation algorithms. Similarly, directory placement and contents influence the processing of queries.

The replication of fragments when they are distributed affects the concurrency control strategies that might be employed. As we will study in Chapter 11, some concurrency control algorithms cannot be easily used with replicated databases. Similarly, usage and access patterns to the database will influence the concurrency control algorithms. If the environment is update intensive, the necessary precautions are quite different from those in a query-only environment.

There is a strong relationship among the concurrency control problem, the deadlock management problem, and reliability issues. This is to be expected, since together they are usually called the *transaction management* problem. The concurrency control algorithm that is employed will determine whether or not a separate deadlock management facility is required. If a locking-based algorithm is used, deadlocks will occur, whereas they will not if timestamping is the chosen alternative.

Reliability mechanisms involve both local recovery techniques and distributed reliability protocols. In that sense, they both influence the choice of the concurrency control techniques and are built on top of them. Techniques to provide reliability also make use of data placement information since the existence of duplicate copies of the data serve as a safeguard to maintain reliable operation.

Finally, the need for replication protocols arise if data distribution involves replicas. As indicated above, there is a strong relationship between replication protocols and concurrency control techniques, since both deal with the consistency of data, but from different perspectives. Furthermore, the replication protocols influence distributed reliability techniques such as commit protocols. In fact, it is sometimes suggested (wrongly, in our view) that replication protocols can be used instead of implementing distributed commit protocols.

1.6.9 Additional Issues

The above design issues cover what may be called "traditional" distributed database systems. The environment has changed significantly since these topics started to be investigated, posing additional challenges and opportunities.

One of the important developments has been the move towards "looser" federation among data sources, which may also be heterogeneous. As we discuss in the next section, this has given rise to the development of multidatabase systems (also called *federated databases* and *data integration systems*) that require re-investigation of some of the fundamental database techniques. These systems constitute an important part of today's distributed environment. We discuss database design issues in multidatabase systems (i.e., *database integration*) in Chapter 4 and the query processing challenges in Chapter 9.

The growth of the Internet as a fundamental networking platform has raised important questions about the assumptions underlying distributed database systems. Two issues are of particular concern to us. One is the re-emergence of peer-to-peer computing, and the other is the development and growth of the World Wide Web (web for short). Both of these aim at improving data sharing, but take different approaches and pose different data management challenges. We discuss peer-to-peer data management in Chapter 16 and web data management in Chapter 17.

We should note that peer-to-peer is not a new concept in distributed databases, as we discuss in the next section. However, their new re-incarnation has significant differences from the earlier versions. In Chapter 16, it is these new versions that we focus on.

Finally, as earlier noted, there is a strong relationship between distributed databases and parallel databases. Although the former assumes each site to be a single logical computer, most of these installations are, in fact, parallel clusters. Thus, while most of the book focuses on issues that arise in managing data distributed across these sites, interesting data management issues exist within a single logical site that may be a parallel system. We discuss these issues in Chapter 14.

1.7 Distributed DBMS Architecture

The architecture of a system defines its structure. This means that the components of the system are identified, the function of each component is specified, and the interrelationships and interactions among these components are defined. The specification of the architecture of a system requires identification of the various modules, with their interfaces and interrelationships, in terms of the data and control flow through the system.

In this section we develop three "reference" architectures[2] for a distributed DBMS: client/server systems, peer-to-peer distributed DBMS, and multidatabase systems. These are "idealized" views of a DBMS in that many of the commercially available systems may deviate from them; however, the architectures will serve as a reasonable framework within which the issues related to distributed DBMS can be discussed.

We first start with a brief presentation of the "ANSI/SPARC architecture", which is a *datalogical* approach to defining a DBMS architecture – it focuses on the different user classes and roles and their varying views on data. This architecture is helpful in putting certain concepts we have discussed so far in their proper perspective. We then have a short discussion of a generic architecture of a centralized DBMSs, that we subsequently extend to identify the set of alternative architectures for a distributed DBMS. Whithin this characterization, we focus on the three alternatives that we identified above.

1.7.1 ANSI/SPARC Architecture

In late 1972, the Computer and Information Processing Committee (X3) of the American National Standards Institute (ANSI) established a Study Group on Database

[2] A reference architecture is commonly created by standards developers to clearly define the interfaces that need to be standardized.

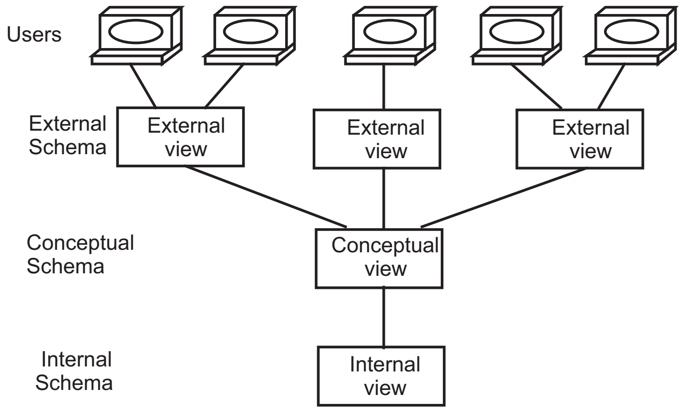

Fig. 1.8 The ANSI/SPARC Architecture

Management Systems under the auspices of its Standards Planning and Requirements Committee (SPARC). The mission of the study group was to study the *feasibility* of setting up standards in this area, as well as determining which aspects should be standardized if it was feasible. The study group issued its interim report in 1975 [ANSI/SPARC, 1975], and its final report in 1977 [Tsichritzis and Klug, 1978]. The architectural framework proposed in these reports came to be known as the "ANSI/SPARC architecture," its full title being "ANSI/X3/SPARC DBMS Framework." The study group proposed that the interfaces be standardized, and defined an architectural framework that contained 43 interfaces, 14 of which would deal with the physical storage subsystem of the computer and therefore not be considered essential parts of the DBMS architecture.

A simplified version of the ANSI/SPARC architecture is depicted in Figure 1.8. There are three views of data: the *external view*, which is that of the end user, who might be a programmer; the *internal view*, that of the system or machine; and the *conceptual view*, that of the enterprise. For each of these views, an appropriate schema definition is required.

At the lowest level of the architecture is the internal view, which deals with the physical definition and organization of data. The location of data on different storage devices and the access mechanisms used to reach and manipulate data are the issues dealt with at this level. At the other extreme is the external view, which is concerned with how users view the database. An individual user's view represents the portion of the database that will be accessed by that user as well as the relationships that the user would like to see among the data. A view can be shared among a number of users, with the collection of user views making up the external schema. In between these two ends is the conceptual schema, which is an abstract definition of the database. It is the "real world" view of the enterprise being modeled in the database [Yormark, 1977]. As such, it is supposed to represent the data and the relationships among data without considering the requirements of individual applications or the restrictions of the physical storage media. In reality, however, it is not possible to ignore these requirements completely, due to performance reasons. The transformation between these three levels is accomplished by mappings that specify how a definition at one level can be obtained from a definition at another level.

This perspective is important, because it provides the basis for data independence that we discussed earlier. The separation of the external schemas from the conceptual schema enables *logical data independence*, while the separation of the conceptual schema from the internal schema allows *physical data independence*.

1.7.2 A Generic Centralized DBMS Architecture

A DBMS is a reentrant program shared by multiple processes (*transactions*), that run database programs. When running on a general purpose computer, a DBMS is interfaced with two other components: the communication subsystem and the operating system. The communication subsystem permits interfacing the DBMS with other subsystems in order to communicate with applications. For example, the terminal monitor needs to communicate with the DBMS to run interactive transactions. The operating system provides the interface between the DBMS and computer resources (processor, memory, disk drives, etc.).

The functions performed by a DBMS can be layered as in Figure 1.9, where the arrows indicate the direction of the data and the control flow. Taking a top-down approach, the layers are the interface, control, compilation, execution, data access, and consistency management.

The *interface layer* manages the interface to the applications. There can be several interfaces such as, in the case of relational DBMSs discussed in Chapter 2, SQL embedded in a host language, such as C and QBE (Query-by-Example). Database application programs are executed against external *views* of the database. For an application, a view is useful in representing its particular perception of the database (shared by many applications). A view in relational DBMSs is a virtual relation derived from base relations by applying relational algebra operations.[3] These concepts are defined more precisely in Chapter 2, but they are usually covered in undergraduate database courses, so we expect many readers to be familiar with them. View management consists of translating the user query from external data to conceptual data.

The *control layer* controls the query by adding semantic integrity predicates and authorization predicates. Semantic integrity constraints and authorizations are usually specified in a declarative language, as discussed in Chapter 5. The output of this layer is an enriched query in the high-level language accepted by the interface.

The *query processing* (or *compilation*) layer maps the query into an optimized sequence of lower-level operations. This layer is concerned with performance. It

[3] Note that this does not mean that the real-world views are, or should be, specified in relational algebra. On the contrary, they are specified by some high-level data language such as SQL. The translation from one of these languages to relational algebra is now well understood, and the effects of the view definition can be specified in terms of relational algebra operations.

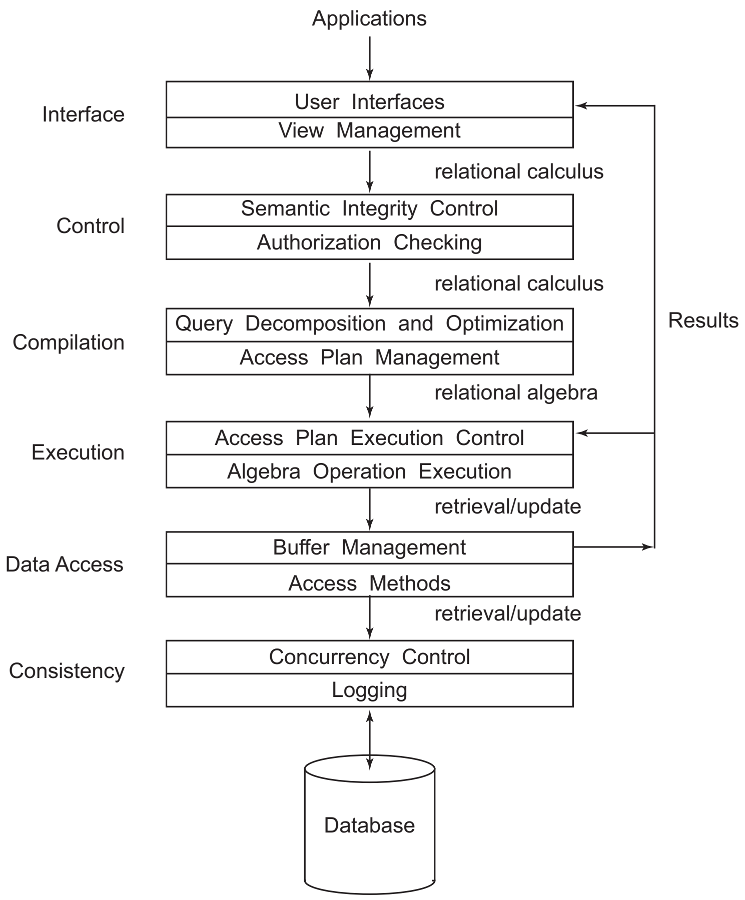

Fig. 1.9 Functional Layers of a Centralized DBMS

decomposes the query into a tree of algebra operations and tries to find the "optimal" ordering of the operations. The result is stored in an access plan. The output of this layer is a query expressed in lower-level code (algebra operations).

The *execution layer* directs the execution of the access plans, including transaction management (commit, restart) and synchronization of algebra operations. It interprets the relational operations by calling the data access layer through the retrieval and update requests.

The *data access layer* manages the data structures that implement the files, indices, etc. It also manages the buffers by caching the most frequently accessed data. Careful use of this layer minimizes the access to disks to get or write data.

Finally, the *consistency layer* manages concurrency control and logging for update requests. This layer allows transaction, system, and media recovery after failure.

1.7.3 Architectural Models for Distributed DBMSs

We now consider the possible ways in which a distributed DBMS may be architected. We use a classification (Figure 1.10) that organizes the systems as characterized with respect to (1) the autonomy of local systems, (2) their distribution, and (3) their heterogeneity.

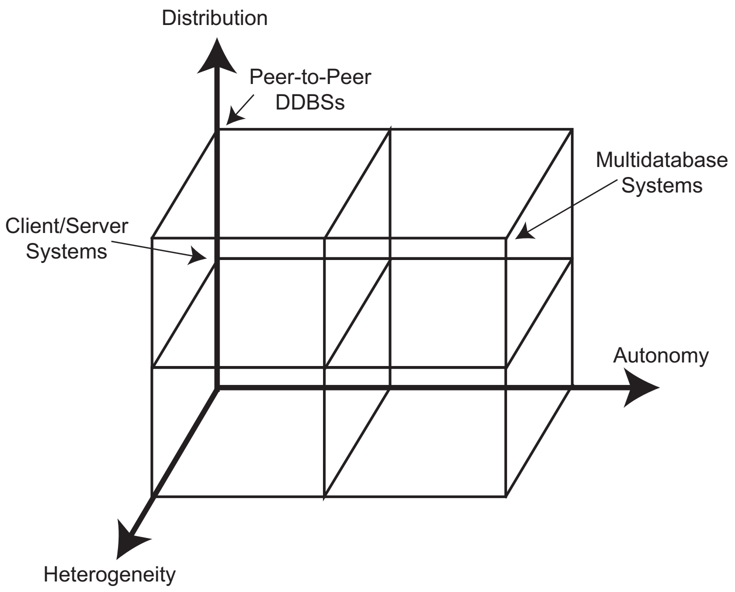

Fig. 1.10 DBMS Implementation Alternatives

1.7.4 Autonomy

Autonomy, in this context, refers to the distribution of control, not of data. It indicates the degree to which individual DBMSs can operate independently. Autonomy is a function of a number of factors such as whether the component systems (i.e., individual DBMSs) exchange information, whether they can independently execute transactions, and whether one is allowed to modify them. Requirements of an autonomous system have been specified as follows [Gligor and Popescu-Zeletin, 1986]:

1. The local operations of the individual DBMSs are not affected by their participation in the distributed system.

2. The manner in which the individual DBMSs process queries and optimize them should not be affected by the execution of global queries that access multiple databases.
3. System consistency or operation should not be compromised when individual DBMSs join or leave the distributed system.

On the other hand, the dimensions of autonomy can be specified as follows [Du and Elmagarmid, 1989]:

1. Design autonomy: Individual DBMSs are free to use the data models and transaction management techniques that they prefer.
2. Communication autonomy: Each of the individual DBMSs is free to make its own decision as to what type of information it wants to provide to the other DBMSs or to the software that controls their global execution.
3. Execution autonomy: Each DBMS can execute the transactions that are submitted to it in any way that it wants to.

We will use a classification that covers the important aspects of these features. One alternative is *tight integration*, where a single-image of the entire database is available to any user who wants to share the information, which may reside in multiple databases. From the users' perspective, the data are logically integrated in one database. In these tightly-integrated systems, the data managers are implemented so that one of them is in control of the processing of each user request even if that request is serviced by more than one data manager. The data managers do not typically operate as independent DBMSs even though they usually have the functionality to do so.

Next we identify *semiautonomous* systems that consist of DBMSs that can (and usually do) operate independently, but have decided to participate in a federation to make their local data sharable. Each of these DBMSs determine what parts of their own database they will make accessible to users of other DBMSs. They are not fully autonomous systems because they need to be modified to enable them to exchange information with one another.

The last alternative that we consider is *total isolation*, where the individual systems are stand-alone DBMSs that know neither of the existence of other DBMSs nor how to communicate with them. In such systems, the processing of user transactions that access multiple databases is especially difficult since there is no global control over the execution of individual DBMSs.

It is important to note at this point that the three alternatives that we consider for autonomous systems are not the only possibilities. We simply highlight the three most popular ones.

1.7.5 Distribution

Whereas autonomy refers to the distribution (or decentralization) of control, the distribution dimension of the taxonomy deals with data. Of course, we are considering the physical distribution of data over multiple sites; as we discussed earlier, the user sees the data as one logical pool. There are a number of ways DBMSs have been distributed. We abstract these alternatives into two classes: *client/server* distribution and *peer-to-peer* distribution (or *full* distribution). Together with the non-distributed option, the taxonomy identifies three alternative architectures.

The client/server distribution concentrates data management duties at servers while the clients focus on providing the application environment including the user interface. The communication duties are shared between the client machines and servers. Client/server DBMSs represent a practical compromise to distributing functionality. There are a variety of ways of structuring them, each providing a different level of distribution. With respect to the framework, we abstract these differences and leave that discussion to Section 1.7.8, which we devote to client/server DBMS architectures. What is important at this point is that the sites on a network are distinguished as "clients" and "servers" and their functionality is different.

In *peer-to-peer systems*, there is no distinction of client machines versus servers. Each machine has full DBMS functionality and can communicate with other machines to execute queries and transactions. Most of the very early work on distributed database systems have assumed peer-to-peer architecture. Therefore, our main focus in this book are on peer-to-peer systems (also called *fully distributed*), even though many of the techniques carry over to client/server systems as well.

1.7.6 Heterogeneity

Heterogeneity may occur in various forms in distributed systems, ranging from hardware heterogeneity and differences in networking protocols to variations in data managers. The important ones from the perspective of this book relate to data models, query languages, and transaction management protocols. Representing data with different modeling tools creates heterogeneity because of the inherent expressive powers and limitations of individual data models. Heterogeneity in query languages not only involves the use of completely different data access paradigms in different data models (set-at-a-time access in relational systems versus record-at-a-time access in some object-oriented systems), but also covers differences in languages even when the individual systems use the same data model. Although SQL is now the standard relational query language, there are many different implementations and every vendor's language has a slightly different flavor (sometimes even different semantics, producing different results).

1.7.7 Architectural Alternatives

The distribution of databases, their possible heterogeneity, and their autonomy are orthogonal issues. Consequently, following the above characterization, there are 18 different possible architectures. Not all of these architectural alternatives that form the design space are meaningful. Furthermore, not all are relevant from the perspective of this book.

In Figure 1.10, we have identified three alternative architectures that are the focus of this book and that we discuss in more detail in the next three subsections: (A0, D1, H0) that corresponds to client/server distributed DBMSs, (A0, D2, H0) that is a peer-to-peer distributed DBMS and (A2, D2, H1) which represents a (peer-to-peer) distributed, heterogeneous multidatabase system. Note that we discuss the heterogeneity issues within the context of one system architecture, although the issue arises in other models as well.

1.7.8 Client/Server Systems

Client/server DBMSs entered the computing scene at the beginning of 1990's and have made a significant impact on both the DBMS technology and the way we do computing. The general idea is very simple and elegant: distinguish the functionality that needs to be provided and divide these functions into two classes: server functions and client functions. This provides a *two-level architecture* which makes it easier to manage the complexity of modern DBMSs and the complexity of distribution.

As with any highly popular term, client/server has been much abused and has come to mean different things. If one takes a process-centric view, then any process that requests the services of another process is its client and vice versa. However, it is important to note that "client/server computing" and "client/server DBMS," as it is used in our context, do not refer to processes, but to actual machines. Thus, we focus on what software should run on the client machines and what software should run on the server machine.

Put this way, the issue is clearer and we can begin to study the differences in client and server functionality. The functionality allocation between clients and serves differ in different types of distributed DBMSs (e.g., relational versus object-oriented). In relational systems, the server does most of the data management work. This means that all of query processing and optimization, transaction management and storage management is done at the server. The client, in addition to the application and the user interface, has a *DBMS client* module that is responsible for managing the data that is cached to the client and (sometimes) managing the transaction locks that may have been cached as well. It is also possible to place consistency checking of user queries at the client side, but this is not common since it requires the replication of the system catalog at the client machines. Of course, there is operating system and communication software that runs on both the client and the server, but we only focus on the DBMS related functionality. This architecture, depicted in Figure 1.11,

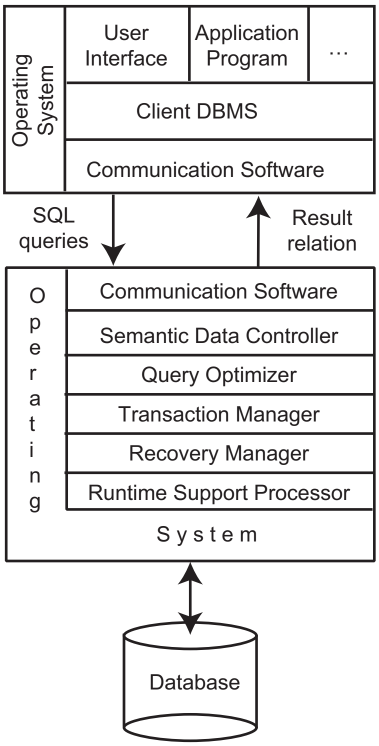

Fig. 1.11 Client/Server Reference Architecture

is quite common in relational systems where the communication between the clients and the server(s) is at the level of SQL statements. In other words, the client passes SQL queries to the server without trying to understand or optimize them. The server does most of the work and returns the result relation to the client.

There are a number of different types of client/server architecture. The simplest is the case where there is only one server which is accessed by multiple clients. We call this *multiple client/single server*. From a data management perspective, this is not much different from centralized databases since the database is stored on only one machine (the server) that also hosts the software to manage it. However, there are some (important) differences from centralized systems in the way transactions are executed and caches are managed. We do not consider such issues at this point. A more sophisticated client/server architecture is one where there are multiple servers in the system (the so-called *multiple client/multiple server* approach). In this case, two alternative management strategies are possible: either each client manages its own connection to the appropriate server or each client knows of only its “home server” which then communicates with other servers as required. The former approach simplifies server code, but loads the client machines with additional responsibilities. This leads to what has been called “heavy client” systems. The latter approach, on the other hand, concentrates the data management functionality at the servers. Thus, the transparency of data access is provided at the server interface, leading to “light clients.”

From a datalogical perspective, client/server DBMSs provide the same view of data as do peer-to-peer systems that we discuss next. That is, they give the user the appearance of a logically single database, while at the physical level data **may** be distributed. Thus the primary distinction between client/server systems and peer-to-peer ones is not in the level of transparency that is provided to the users and applications, but in the architectural paradigm that is used to realize this level of transparency.

Client/server can be naturally extended to provide for a more efficient function distribution on different kinds of servers: *client servers* run the user interface (e.g., web servers), *application servers* run application programs, and *database servers* run database management functions. This leads to the present trend in three-tier distributed system architecture, where sites are organized as specialized servers rather than as general-purpose computers.

The original idea, which is to offload the database management functions to a special server, dates back to the early 1970s [Canaday et al., 1974]. At the time, the computer on which the database system was run was called the *database machine*, *database computer*, or *backend computer*, while the computer that ran the applications was called the *host computer*. More recent terms for these are the *database server* and *application server*, respectively. Figure 1.12 illustrates a simple view of the database server approach, with application servers connected to one database server via a communication network.

The database server approach, as an extension of the classical client/server architecture, has several potential advantages. First, the single focus on data management makes possible the development of specific techniques for increasing data reliability and availability, e.g. using parallelism. Second, the overall performance of database management can be significantly enhanced by the tight integration of the database system and a dedicated database operating system. Finally, a database server can also exploit recent hardware architectures, such as multiprocessors or clusters of PC servers to enhance both performance and data availability.

Although these advantages are significant, they can be offset by the overhead introduced by the additional communication between the application and the data servers. This is an issue, of course, in classical client/server systems as well, but in this case there is an additional layer of communication to worry about. The communication cost can be amortized only if the server interface is sufficiently high level to allow the expression of complex queries involving intensive data processing.

The application server approach (indeed, a n-tier distributed approach) can be extended by the introduction of multiple database servers and multiple application servers (Figure 1.13), as can be done in classical client/server architectures. In this case, it is typically the case that each application server is dedicated to one or a few applications, while database servers operate in the multiple server fashion discussed above.

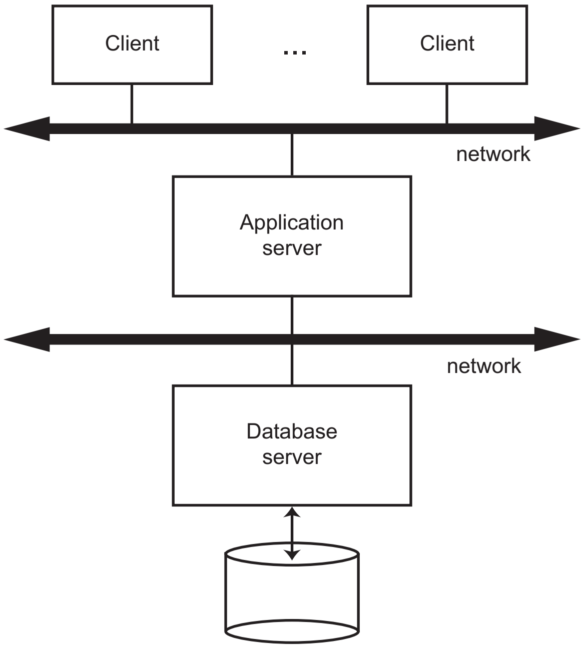

Fig. 1.12 Database Server Approach

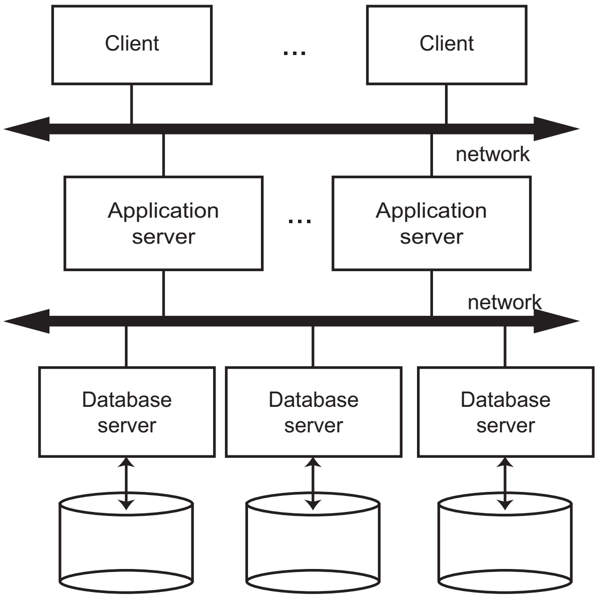

Fig. 1.13 Distributed Database Servers

1.7.9 Peer-to-Peer Systems

If the term “client/server” is loaded with different interpretations, “peer-to-peer” is even worse as its meaning has changed and evolved over the years. As noted earlier, the early works on distributed DBMSs all focused on peer-to-peer architectures where there was no differentiation between the functionality of each site in the system[4]. After a decade of popularity of client/server computing, peer-to-peer have made a comeback in the last few years (primarily spurred by file sharing applications) and some have even positioned peer-to-peer data management as an alternative to distributed DBMSs. While this may be a stretch, modern peer-to-peer systems have two important differences from their earlier relatives. The first is the massive distribution in current systems. While in the early days we focused on a few (perhaps at most tens of) sites, current systems consider thousands of sites. The second is the inherent heterogeneity of every aspect of the sites and their autonomy. While this has always been a concern of distributed databases, as discussed earlier, coupled with massive distribution, site heterogeneity and autonomy take on an added significance, disallowing some of the approaches from consideration.

Discussing peer-to-peer database systems within this backdrop poses real challenges; the unique issues of database management over the “modern” peer-to-peer architectures are still being investigated. What we choose to do, in this book, is to initially focus on the classical meaning of peer-to-peer (the same functionality of each site), since the principles and fundamental techniques of these systems are very similar to those of client/server systems, and discuss the modern peer-to-peer database issues in a separate chapter (Chapter 16).

Let us start the description of the architecture by looking at the data organizational view. We first note that the physical data organization on each machine may be, and probably is, different. This means that there needs to be an individual internal schema definition at each site, which we call the *local internal schema* (LIS). The enterprise view of the data is described by the *global conceptual schema* (GCS), which is global because it describes the logical structure of the data at all the sites.

To handle data fragmentation and replication, the logical organization of data at each site needs to be described. Therefore, there needs to be a third layer in the architecture, the *local conceptual schema* (LCS). In the architectural model we have chosen, then, the global conceptual schema is the union of the local conceptual schemas. Finally, user applications and user access to the database is supported by *external schemas* (ESs), defined as being above the global conceptual schema.

This architecture model, depicted in Figure 1.14, provides the levels of transparency discussed earlier. Data independence is supported since the model is an extension of ANSI/SPARC, which provides such independence naturally. Location and replication transparencies are supported by the definition of the local and global conceptual schemas and the mapping in between. Network transparency, on the other hand, is supported by the definition of the global conceptual schema. The user

[4] In fact, in the first edition of this book which appeared in early 1990 and whose writing was completed in 1989, there wasn’t a single mention of the term “client/server”.

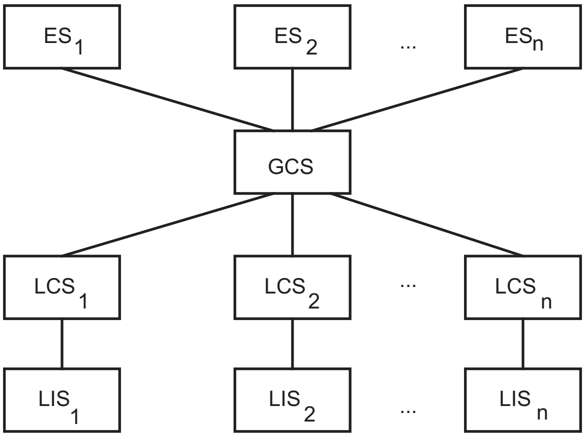

Fig. 1.14 Distributed Database Reference Architecture

queries data irrespective of its location or of which local component of the distributed database system will service it. As mentioned before, the distributed DBMS translates global queries into a group of local queries, which are executed by distributed DBMS components at different sites that communicate with one another.

The detailed components of a distributed DBMS are shown in Figure 1.15. One component handles the interaction with users, and another deals with the storage. The first major component, which we call the *user processor*, consists of four elements:

1. The *user interface handler* is responsible for interpreting user commands as they come in, and formatting the result data as it is sent to the user.
2. The *semantic data controller* uses the integrity constraints and authorizations that are defined as part of the global conceptual schema to check if the user query can be processed. This component, which is studied in detail in Chapter 5, is also responsible for authorization and other functions.
3. The *global query optimizer and decomposer* determines an execution strategy to minimize a cost function, and translates the global queries into local ones using the global and local conceptual schemas as well as the global directory. The global query optimizer is responsible, among other things, for generating the best strategy to execute distributed join operations. These issues are discussed in Chapters 6 through 8.
4. The *distributed execution monitor* coordinates the distributed execution of the user request. The execution monitor is also called the *distributed transaction manager*. In executing queries in a distributed fashion, the execution monitors at various sites may, and usually do, communicate with one another.

The second major component of a distributed DBMS is the *data processor* and consists of three elements:

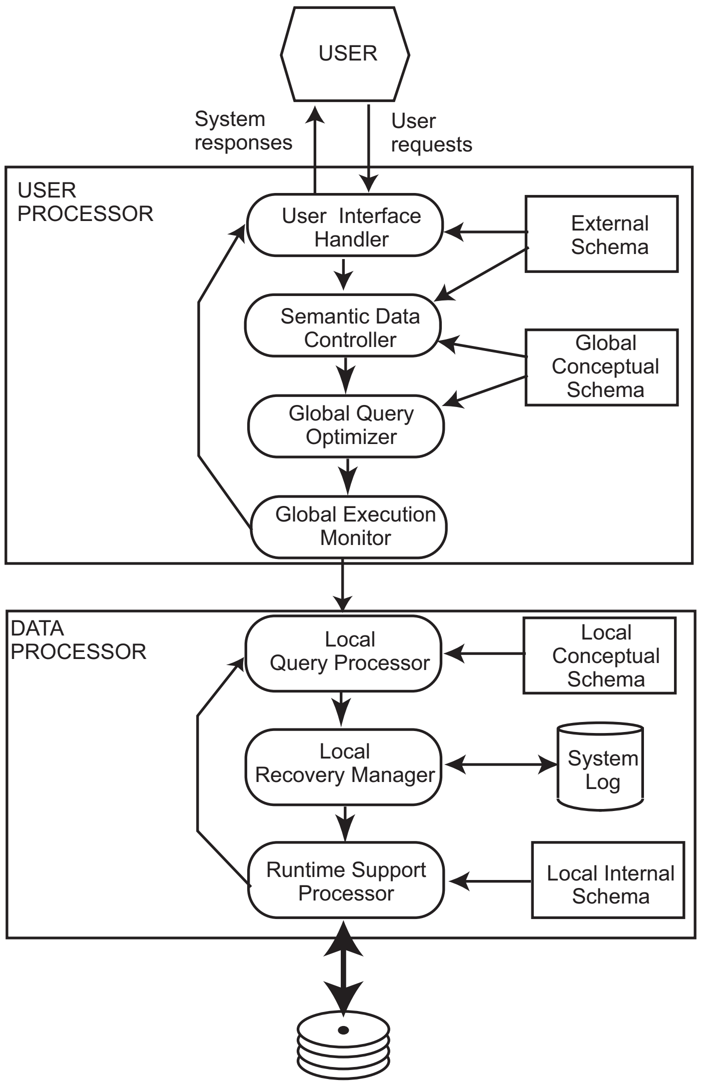

Fig. 1.15 Components of a Distributed DBMS

1. The *local query optimizer*, which actually acts as the *access path selector*, is responsible for choosing the best access path[5] to access any data item (touched upon briefly in Chapter 8).
2. The *local recovery manager* is responsible for making sure that the local database remains consistent even when failures occur (Chapter 12).
3. The *run-time support processor* physically accesses the database according to the physical commands in the schedule generated by the query optimizer. The run-time support processor is the interface to the operating system and contains the *database buffer* (or *cache*) *manager*, which is responsible for maintaining the main memory buffers and managing the data accesses.

It is important to note, at this point, that our use of the terms "user processor" and "data processor" does not imply a functional division similar to client/server systems. These divisions are merely organizational and there is no suggestion that they should be placed on different machines. In peer-to-peer systems, one expects to find both the user processor modules and the data processor modules on each machine. However, there have been suggestions to separate "query-only sites" in a system from full-functionality ones. In this case, the former sites would only need to have the user processor.

In client/server systems where there is a single server, the client has the user interface manager while the server has all of the data processor functionality as well as semantic data controller; there is no need for the global query optimizer or the global execution monitor. If there are multiple servers and the home server approach described in the previous section is employed, then each server hosts all of the modules except the user interface manager that resides on the client. If, however, each client is expected to contact individual servers on its own, then, most likely, the clients will host the full user processor functionality while the data processor functionality resides in the servers.

1.7.10 Multidatabase System Architecture

Multidatabase systems (MDBS) represent the case where individual DBMSs (whether distributed or not) are fully autonomous and have no concept of cooperation; they may not even "know" of each other's existence or how to talk to each other. Our focus is, naturally, on distributed MDBSs, which is what the term will refer to in the remainder. In most current literature, one finds the term *data integration system* used instead. We avoid using that term since data integration systems consider non-database data sources as well. Our focus is strictly on databases. We discuss these systems and their relationship to database integration in Chapter 4. We note, however, that there is considerable variability of the use of the term "multidatabase" in literature. In this

[5] The term *access path* refers to the data structures and the algorithms that are used to access the data. A typical access path, for example, is an index on one or more attributes of a relation.

book, we use it consistently as defined above, which may devitate from its use in some of the existing literature.

The differences in the level of autonomy between the distributed multi-DBMSs and distributed DBMSs are also reflected in their architectural models. The fundamental difference relates to the definition of the global conceptual schema. In the case of logically integrated distributed DBMSs, the global conceptual schema defines the conceptual view of the *entire* database, while in the case of distributed multi-DBMSs, it represents only the collection of *some* of the local databases that each local DBMS wants to share. The individual DBMSs may choose to make some of their data available for access by others (i.e., federated database architectures) by defining an *export schema* [Heimbigner and McLeod, 1985]. Thus the definition of a *global database* is different in MDBSs than in distributed DBMSs. In the latter, the global database is equal to the union of local databases, whereas in the former it is only a (possibly proper) subset of the same union. In a MDBS, the GCS (which is also called a mediated *schema*) is defined by integrating either the external schemas of local autonomous databases or (possibly parts of their) local conceptual schemas.

Furthermore, users of a local DBMS define their own views on the local database and do not need to change their applications if they do not want to access data from another database. This is again an issue of autonomy.

Designing the global conceptual schema in multidatabase systems involves the integration of either the local conceptual schemas or the local external schemas (Figure 1.16). A major difference between the design of the GCS in multi-DBMSs and in logically integrated distributed DBMSs is that in the former the mapping is from local conceptual schemas to a global schema. In the latter, however, mapping is in the reverse direction. As we discuss in Chapters 3 and 4, this is because the design in the former is usually a bottom-up process, whereas in the latter it is usually a top-down procedure. Furthermore, if heterogeneity exists in the multidatabase system, a canonical data model has to be found to define the GCS.

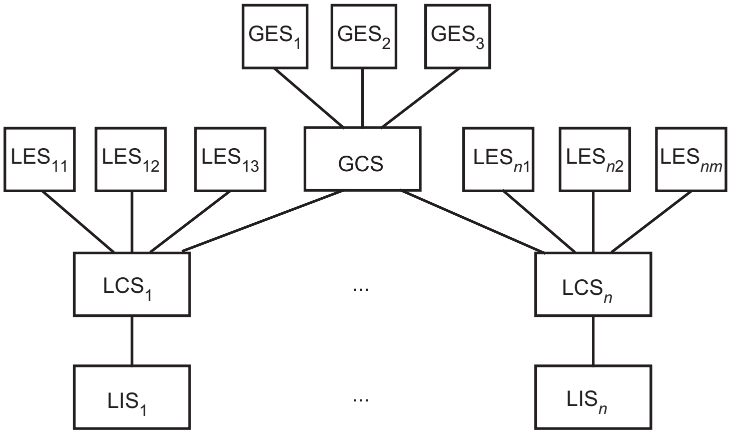

Fig. 1.16 MDBS Architecture with a GCS

Once the GCS has been designed, views over the global schema can be defined for users who require global access. It is not necessary for the GES and GCS to be defined using the same data model and language; whether they do or not determines whether the system is homogeneous or heterogeneous.

If heterogeneity exists in the system, then two implementation alternatives exist: unilingual and multilingual. A *unilingual* multi-DBMS requires the users to utilize possibly different data models and languages when both a local database and the global database are accessed. The identifying characteristic of unilingual systems is that any application that accesses data from multiple databases must do so by means of an external view that is defined on the global conceptual schema. This means that the user of the global database is effectively a different user than those who access only a local database, utilizing a different data model and a different data language.

An alternative is *multilingual* architecture, where the basic philosophy is to permit each user to access the global database (i.e., data from other databases) by means of an external schema, defined using the language of the user's local DBMS. The GCS definition is quite similar in the multilingual architecture and the unilingual approach, the major difference being the definition of the external schemas, which are described in the language of the external schemas of the local database. Assuming that the definition is purely local, a query issued according to a particular schema is handled exactly as any query in the centralized DBMSs. Queries against the global database are made using the language of the local DBMS, but they generally require some processing to be mapped to the global conceptual schema.

The component-based architectural model of a multi-DBMS is significantly different from a distributed DBMS. The fundamental difference is the existence of full-fledged DBMSs, each of which manages a different database. The MDBS provides a layer of software that runs on top of these individual DBMSs and provides users with the facilities of accessing various databases (Figure 1.17). Note that in a distributed MDBS, the multi-DBMS layer may run on multiple sites or there may be central site where those services are offered. Also note that as far as the individual DBMSs are concerned, the MDBS layer is simply another application that submits requests and receives answers.

A popular implementation architecture for MDBSs is the mediator/wrapper approach (Figure 1.18) [Wiederhold, 1992]. A *mediator* "is a software module that exploits encoded knowledge about certain sets or subsets of data to create information for a higher layer of applications." Thus, each mediator performs a particular function with clearly defined interfaces. Using this architecture to implement a MDBS, each module in the multi-DBMS layer of Figure 1.17 is realized as a mediator. Since mediators can be built on top of other mediators, it is possible to construct a layered implementation. In mapping this architecture to the datalogical view of Figure 1.16, the mediator level implements the GCS. It is this level that handles user queries over the GCS and performs the MDBS functionality.

The mediators typically operate using a common data model and interface language. To deal with potential heterogeneities of the source DBMSs, *wrappers* are implemented whose task is to provide a mapping between a source DBMSs view and the mediators' view. For example, if the source DBMS is a relational one, but the

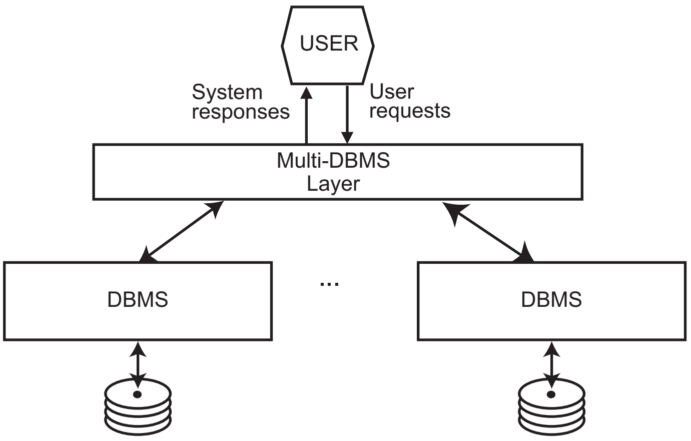

Fig. 1.17 Components of an MDBS

mediator implementations are object-oriented, the required mappings are established by the wrappers. The exact role and function of mediators differ from one implementation to another. In some cases, thin mediators have been implemented who do nothing more than translation. In other cases, wrappers take over the execution of some of the query functionality.

One can view the collection of mediators as a middleware layer that provides services above the source systems. Middleware is a topic that has been the subject of significant study in the past decade and very sophisticated middleware systems have been developed that provide advanced services for development of distributed applications. The mediators that we discuss only represent a subset of the functionality provided by these systems.

1.8 Bibliographic Notes

There are not many books on distributed DBMSs. Ceri and Pelagatti's book [Ceri and Pelagatti, 1983] was the first on this topic though it is now dated. The book by Bell and Grimson [Bell and Grimson, 1992] also provides an overview of the topics addressed here. In addition, almost every database book now has a chapter on distributed DBMSs. A brief overview of the technology is provided in [Özsu and Valduriez, 1997]. Our papers [Özsu and Valduriez, 1994, 1991] provide discussions of the state-of-the-art at the time they were written.

Database design is discussed in an introductory manner in [Levin and Morgan, 1975] and more comprehensively in [Ceri et al., 1987]. A survey of the file distribution algorithms is given in [Dowdy and Foster, 1982]. Directory management has not been considered in detail in the research community, but general techniques can be found in Chu and Nahouraii [1975] and [Chu, 1976]. A survey of query processing

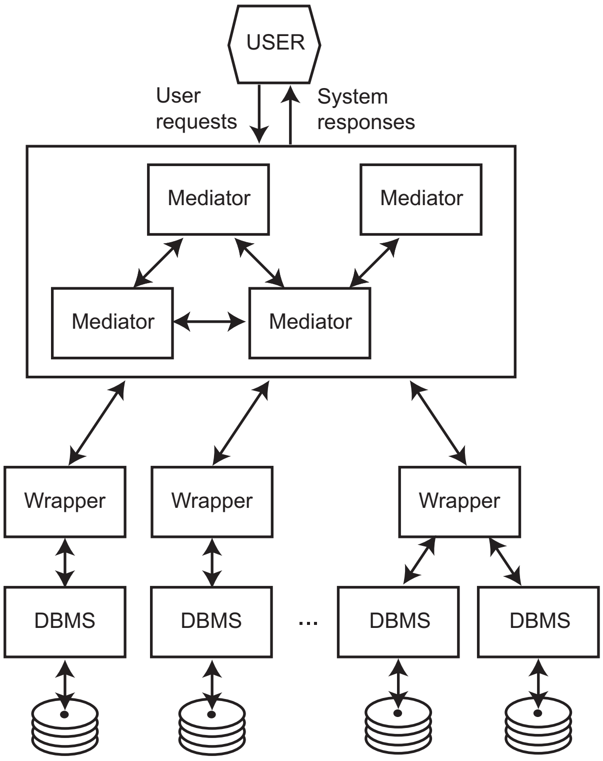

Fig. 1.18 Mediator/Wrapper Architecture

techniques can be found in [Sacco and Yao, 1982]. Concurrency control algorithms are reviewed in [Bernstein and Goodman, 1981] and [Bernstein et al., 1987]. Deadlock management has also been the subject of extensive research; an introductory paper is [Isloor and Marsland, 1980] and a widely quoted paper is [Obermarck, 1982]. For deadlock detection, good surveys are [Knapp, 1987] and [Elmagarmid, 1986]. Reliability is one of the issues discussed in [Gray, 1979], which is one of the landmark papers in the field. Other important papers on this topic are [Verhofstadt, 1978] and [Härder and Reuter, 1983]. [Gray, 1979] is also the first paper discussing the issues of operating system support for distributed databases; the same topic is addressed in [Stonebraker, 1981]. Unfortunately, both papers emphasize centralized database systems.

There have been a number of architectural framework proposals. Some of the interesting ones include Schreiber's quite detailed extension of the ANSI/SPARC framework which attempts to accommodate heterogeneity of the data models [Schreiber, 1977], and the proposal by Mohan and Yeh [Mohan and Yeh, 1978]. As expected, these date back to the early days of the introduction of distributed DBMS technology. The detailed component-wise system architecture given in Figure 1.15 derives from

[Rahimi, 1987]. An alternative to the classification that we provide in Figure 1.10 can be found in [Sheth and Larson, 1990].

Most of the discussion on architectural models for multi-DBMSs is from [Özsu and Barker, 1990]. Other architectural discussions on multi-DBMSs are given in [Gligor and Luckenbaugh, 1984], [Litwin, 1988], and [Sheth and Larson, 1990]. All of these papers provide overview discussions of various prototype and commercial systems. An excellent overview of heterogeneous and federated database systems is [Sheth and Larson, 1990].